Computer Vision 2023–2024

Cristian Perez Jensen

January 12, 2025

Note that these are not the official lecture notes of the course, but only notes written by a student of the course. As such, there might be mistakes. The source code can be found at `github.com/cristianpjensen/eth-cs-notes`. If you find a mistake, please create an issue or open a pull request.

Contents

List of symbols

$\doteq$	Equality by definition
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i : j$	Set of natural numbers between i and j. I.e., $\{i, i+1, \ldots, j\}$
$f : A \to B$	Function f that maps elements of set A to elements of set B
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)

$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	Tensor
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$

$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}}^2 f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$

$\boldsymbol{\theta} \in \Theta$	Parametrization of a model, where Θ is a compact subset of $\mathbb{R}^K$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	Labeled training data

x	2-dimensional vector in real space $\mathbb{R}$
$\tilde{x}$	2-dimensional vector in projective space $\mathbb{P}$
$\overline{x}$	2-dimensional vector in projective space $\mathbb{P}$ with $w = 1$
X	3-dimensional vector in real space $\mathbb{R}$
$\tilde{X}$	3-dimensional vector in projective space $\mathbb{P}$
$\overline{X}$	3-dimensional vector in projective space $\mathbb{P}$ with $w = 1$

1 Local features

For many computer vision tasks, we need to be able to identify distinctive points that can be matched to points of other images.[1] These tasks require that points that depict the same object must be detected independently in different images, and it must be possible to match such corresponding points between images. This leads to the following requirements,

- Translation, rotation, and scale invariance;

- Robustness to affine transformations, noise, and occlusion;

- A sufficient amount of points have to be detected over an object;

- Regions must contain distinctive structure;

- Real-time performance.

The following is a general approach for finding and matching keypoints between images,

1. Find a set of distinctive keypoints;

2. Define a region around each keypoint, dependent on scale;

3. Extract and normalize the region content to make it invariant to affine transformations and noise;

4. Compute a local descriptor from the normalized region;

5. Match local descriptors between images.

1.1 Harris detector

The *Harris detector* is an algorithm for keypoint localization [Harris et al., 1988]. We are looking for points that have significant change in both the x- and y-direction, *i.e.* corners, such that the point will be localizable at any orientation. A good illustration of this is Figure 1.1.

To find corners, we want to find points such that if we shift the window in any direction, it should give a large change in intensity. Let's define an error function as the following,

$$E(u,v) \doteq \sum_{(x,y)\in W} (\mathbf{I}[x+u,y+v] - \mathbf{I}[x,y])^2,$$

where W is the set of all points in the window that is a potential corner. We want to find Ws such that $E(u,v)$ is large for any (u,v). We can rederive $E(u,v)$ with its first-order Taylor expansion to make this easy
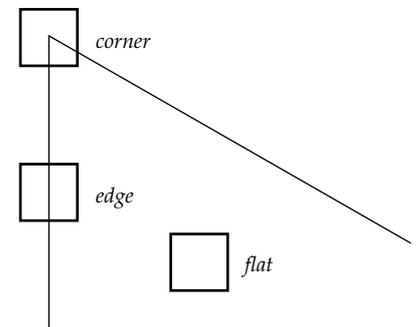
Figure 1.1. The Harris detector seeks to find corner points, which have a significant change in both the x- and y-direction. Edges are not localizable, because if we rotate the patch, we will not be able to match the two rotated patches together.

and efficient:

$$E(u,v) \doteq \sum_{(x,y) \in W} (\mathbf{I}[x+u, y+v] - \mathbf{I}[x,y])^2$$

$$\approx \sum_{(x,y) \in W} \left(\mathbf{I}[x,y] + \begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} - \mathbf{I}[x,y] \right)^2$$

$$= \sum_{(x,y) \in W} \left(\begin{bmatrix} \mathbf{I}_x & \mathbf{I}_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \right)^2$$

$$= \begin{bmatrix} u & v \end{bmatrix} \mathbf{M} \begin{bmatrix} u \\ v \end{bmatrix}$$

First-order Taylor approximation

where $\mathbf{I}_x$ and $\mathbf{I}_y$ are the gradients in the x- and y-direction at the implicit $[x,y]$ point, and $\mathbf{M}$ is the structure tensor,

$$\mathbf{M} = \sum_{(x,y) \in W} \begin{bmatrix} \mathbf{I}_x^2 & \mathbf{I}_x \mathbf{I}_y \\ \mathbf{I}_x \mathbf{I}_y & \mathbf{I}_y^2 \end{bmatrix}.$$

This can be computed using convolutions, which makes it efficient to compute for all pixels at once.

We want to find points with an error surface as in Figure 1.1c. The eigenvalues of a matrix tell us how much it changes in the direction of their eigenvectors. Since we found that $E(u,v)$ is approximately equal to a matrix multiplication with $\mathbf{M}$, the eigenvalues of $\mathbf{M}$ tell us how much the error function changes as we shift the window in the direction of the eigenvectors. Thus, we want to find points where both eigenvalues are large. The Harris detector uses the following response function, because it can be rederived to not need the (computationally inefficient) eigendecomposition,

$$R = \lambda_1 \lambda_2 - \kappa (\lambda_1 + \lambda_2)^2$$
$$= \det(\mathbf{M}) - \kappa \operatorname{tr}(\mathbf{M})^2.$$

The response function R is a measure of a window's "cornerness". If $R > 0$, its window is considered to be a corner for the Harris detector.
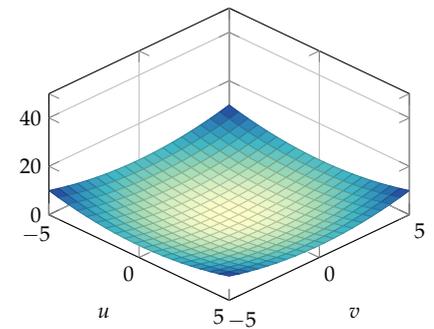
Lastly, the Harris detector applies non-maximum suppression, which reduces clusters of points with $R > 0$ to a single point, since they capture the same corner.

1.2 Scale-invariant region selection

The Harris detector is only able to find keypoints at a fixed scale. However, we would like to be able to detect keypoints at any scale and use this scale to match keypoints of different scales. In other words, we want to be able to match keypoints in a zoomed in version of an image to points in a zoomed out one. The naive approach would be to compare descriptors while varying the patch size, but this results in a combinatorial search problem.

(a) Error surface of a patch containing a flat area.
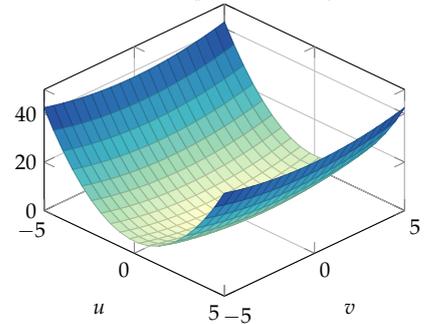
(b) Error surface of a patch containing an edge.

(c) Error surface of a patch containing a corner.

Figure 1.2. Different types of error surfaces.

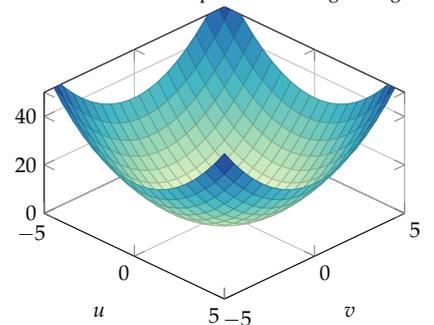A better idea is to do detection over a scale space, which has three dimensions x, y, σ, where the σ parametrizes the convolution with a Laplacian of Gaussian (second derivative of the Gaussian). This "blurs" the image so only large structures remain. Then, we compute the response function for all points (x, y, σ), and call points that are a maximum in a $3 \times 3 \times 3$ grid around them a feature point. The Laplacian of Gaussian can be approximated by the difference of Gaussian operator, which is more efficient.

1.3 Local descriptors

SIFT is an algorithm that uses the orientation of gradients within a patch to compute a keypoint descriptor [Lowe, 2004]. It does so by dividing the found patch into a 4×4 grid and computing the 8-bin histogram of gradient orientations, weighted by their magnitude, within each grid cell. This results in a $4 \times 4 \times 8 = 128$-dimensional vector representation of a keypoint.

We can assign an orientation to a keypoint by creating a histogram of local gradient orientations, weighted by a Gaussian window, within the patch and assigning the orientation to be the maximum gradient orientation. We can then align the descriptor to this orientation, such that we can directly compare two descriptors.

1.4 Matching

There are three main ways of matching keypoints between two images,

- *One-way matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference;

- *Mutual matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference only if this condition also holds the other way;

- *Ratio-threshold matching*: for each keypoint in image 1, match it to the keypoint in image 2 with the smallest descriptor difference only if the ratio between the smallest and second smallest difference is less than some threshold.
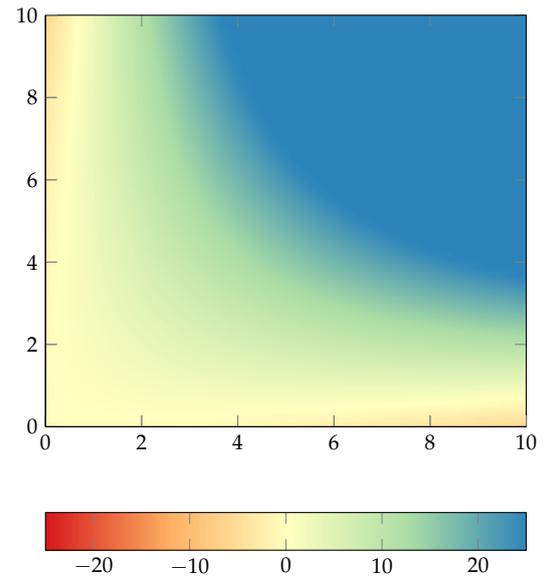
Figure 1.3. Harris detector response function for $\kappa = 0.06$. λ_1 is shown on the x-axis, and λ_2 is shown on the y-axis.

2 Deep learning

In this course, we will see many computer vision tasks that pre-date the deep learning era. However, these tasks have seen much improvement by applying machine learning to them. Thus, we will first be introduced to the tasks, followed by how the problem was dealt with initially, and then how machine learning is applied to it.

2.1 Convolutional neural networks

TODO

2.2 Sequence models

A *recurrent neural network* (RNN) is a neural network that maps from an input space of sequences to an output space of sequences in a stateful way [Rumelhart et al., 1985]. That is, the prediction of the output y_t depends on all inputs x that came before t. This is done by keeping track of a hidden state h_t, which is a function of all previous inputs and the current input,

$$h_t = \sigma(W_h h_{t-1} + W_x x_t).$$
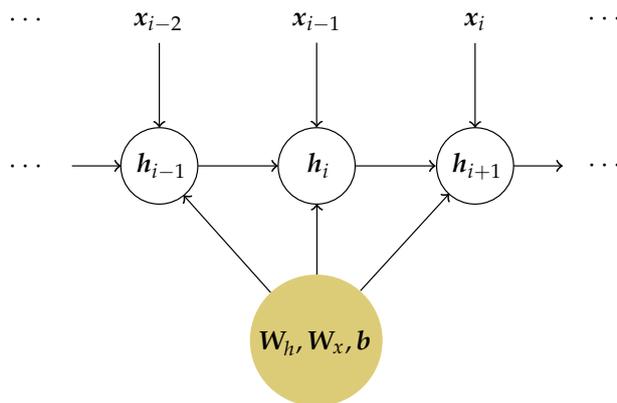
An application of RNNs in computer vision is processing video, where each frame is processed by a CNN. Then, each frame's vector representation is used to update the hidden state of the RNN to obtain a representation of the video.

Figure 2.1. Computation graph of an RNN.

Putting everything together, backpropagation computes the following

gradient w.r.t. W_h,

$$\frac{\partial \ell}{\partial W_h} = \sum_{j=0}^{T} \frac{\partial \ell_j}{\partial W_h}$$

$$= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial h_k} \frac{\partial h_k}{\partial W_h}$$

$$= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k} \frac{\partial h_k}{\partial W_h}$$

$$= \sum_{j=0}^{T} \sum_{k=1}^{j} \frac{\partial \ell_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial W_h},$$

where T is the amount of timesteps in the sequence. If we use the following approximation $\frac{\partial h_m}{\partial h_{m-1}} \approx W_h$, then we see that there are a large amount of matrix multiplications of W_h with itself. This will lead to either vanishing or exploding gradients, depending on whether $\det(W_h)$ is greater or less than 1. Possible solutions for this are gradient clipping and gated recurrent units [Hochreiter and Schmidhuber, 1997, Cho et al., 2014].

2.3 Transformers

Attention is a mechanism in neural networks that a model can learn to make predictions by selectively attending to a given set of data by using query q, key k, and value v vector representations. The query and key vectors are used to determine how much weight should be given to the value vector.[2] The weights are computed by $A_i = \text{softmax}(q_i^\top k_i)$, so the values after the attention block can be computed as follows,

$$\text{att}(x_i) = \sum_j A_{ij} v_i.$$

Self-attention blocks learn the query, key, and value representations from data. More specifically, it learns matrices W_Q, W_K, and W_V and computes the vectors from these matrices,

$$Q = W_Q^\top X$$
$$K = W_K^\top X$$
$$V = W_V^\top X.$$

Then, we can use these to compute the output of the self-attention block,

$$\text{selfatt}(X) = \text{softmax}\left(\frac{Q^\top K}{\sqrt{d_q}} \right) V,$$

where d_q is the dimensionality of the query and key vectors. Furthermore, we need to add a positional encoding to provide ordering information
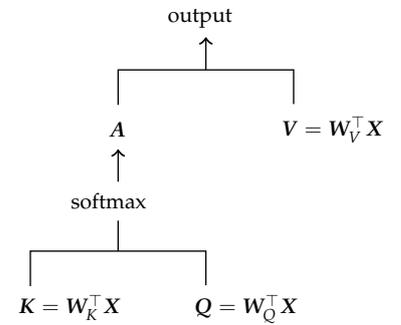
Figure 2.2. Self-attention mechanism.

[2] Note the parallel with dictionaries/hashmaps in programming languages, but, in the attention mechanism, we do a "soft-lookup".

output

layer norm

$\oplus$

MLPs

layer norm

$\oplus$

multi-head self-attention

residual connections

$\oplus$ *positional embeddings*

X

Figure 2.3. Transformer encoder architecture.

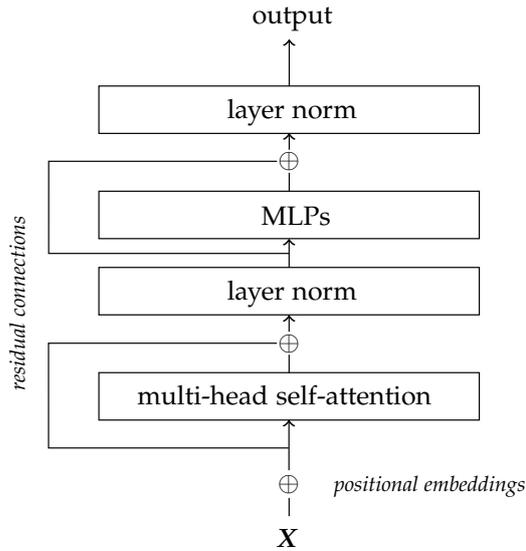to the model, because the self-attention operation is permutation equivariant. This is done by a sinusoidal positional encoding and are simply combined with X by addition.

Transformers use multi-headed self-attention, which is a module where self-attention is applied M times independently to the data. Thus, this module learns M different ways of looking at the same dataset. The outputs of each self-attention block is concatenated and linearly transformed to the expected dimensionality. Transformers follow this by normalization and MLP layers, as can be seen in Figure 2.3.

2.4 Latent variable models

Latent variable models map between an observation space $x \in \mathbb{R}^D$ and latent space $z \in \mathbb{R}^Q$ with $Q < D$. The models then learn two functions: encoder $f_\theta : \mathbb{R}^D \to \mathbb{R}^Q$ and decoder $g_\theta : \mathbb{R}^Q \to \mathbb{R}^D$. These models are called *autoencoders*, because they predict their input as output.

Generative latent variable models are Bayesian probabilistic functions,

$$p(x) = \int_z p(x \mid z)p(z)dz.$$

So, the probability of a sample is the "weighted average" of this sample given a latent variable weighted by the probability of this latent variable. Generally, the goal is to be able to sample from this probability distribution.

Variational autoencoders (VAE) sample z from a simple distributions such as the Gaussian and learn decoders that map from this distribution to data points x [Kingma and Welling, 2013]. The aim of training is to

maximize the probability of each x in the training set, *i.e.*, maximize the likelihood of seeing our data. Since the likelihood (also known as evidence) is an intractable integral, we need to derive a tractable lower bound,

$$
\begin{aligned}
\log p(x) &= \log \int p(x \mid z) p(z) dz \\
&= \log \int p(x \mid z) \frac{p(z)}{q(z)} q(z) dz \\
&\geq \int q(z) \log \left(p(x \mid z) \frac{p(z)}{q(z)} \right) dz \quad\quad \text{Jensen's inequality} \\
&= \mathbb{E}_q \left[\log \frac{p(x \mid z) p(z)}{q(z)} \right],
\end{aligned}
$$

which is called the *evidence lower bound* (ELBO). The probability distribution q_θ is the encoder and can be defined to be anything, *e.g.*, a Gaussian distribution $\mathcal{N}(0, I)$ or a probabilistic deep learning model q_θ dependent on x.

An objective function can be derived from the ELBO, which maximizes the lower bound of the likelihood,

$$
\begin{aligned}
\mathbb{E}_q \left[\log \frac{p(x \mid z) p(z)}{q(z)} \right] &= \mathbb{E}_q[\log p(x \mid z)] - \mathbb{E}_q \left[-\log \frac{p(z)}{q(z)} \right] \\
&= \mathbb{E}_q[\log p(x \mid z)] - D_{\mathrm{KL}}(q \parallel p).
\end{aligned}
$$

The negative log-likelihood is thus upper-bounded by

$$
D_{\mathrm{KL}}(q \parallel p) + \mathbb{E}_q[-\log p(x \mid z)].
$$

VAEs minimize this bound,

$$
\hat{\theta} = \underset{\theta \in \Theta}{\operatorname{argmin}} \sum_{x \in \mathcal{D}} \underbrace{D_{\mathrm{KL}}(q_\theta(z \mid x) \parallel p(z))}_{\text{encoder loss}} + \underbrace{\mathbb{E}_{q_\theta(z \mid x)}[-\log p_\theta(x \mid z)]}_{\text{decoder loss}},
$$

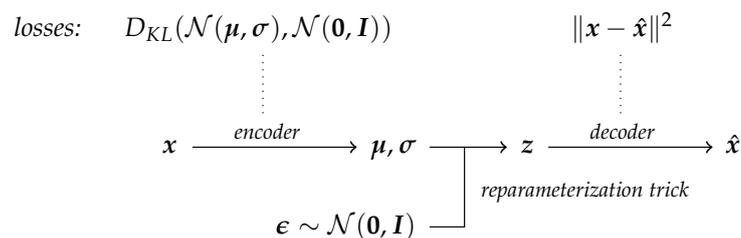where we now learn a latent representation z given the datapoint x.


Figure 2.4. Variational autoencoder architecture.

Because of problems with the gradient, the encoder does not directly predict z from x, but rather predicts a Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ from x, which is used to sample z. This splits the model into an encoder

and a decoder network that are updated separately, as can be seen in Figure 2.4.

Generative adversarial networks (GAN) do not explicitly model the likelihood. Instead, they use an adversarial process in which two models are trained simultaneously: a generator $G : \mathbb{R}^Q \to \mathbb{R}^D$ and a discriminator $D : \mathbb{R}^D \to [0, 1]$ [Goodfellow et al., 2014]. The generator captures the data distribution, while the discriminator estimates whether a sample comes from the data distribution or the generator. At a high level, we train D to assign probability 1 to samples from the training data and 0 to samples from the generator, while training G to fool D such that it assigns probability 1 to its samples,

$$\underset{\boldsymbol{\theta}_G}{\operatorname{argmin}} \, \underset{\boldsymbol{\theta}_D}{\operatorname{argmax}} \, \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})} \left[\log D_{\boldsymbol{\theta}_D}(\boldsymbol{x}) \right] + \mathbb{E}_{p(\boldsymbol{z})} \left[\log(1 - D_{\boldsymbol{\theta}_D}(G_{\boldsymbol{\theta}_G}(\boldsymbol{z}))) \right].$$

The loss function of G solely depends on D, which is learned. So, GANs are a way of learning the loss function.

generator $\longrightarrow \hat{x} \xrightarrow{\textit{discriminator}} 0$
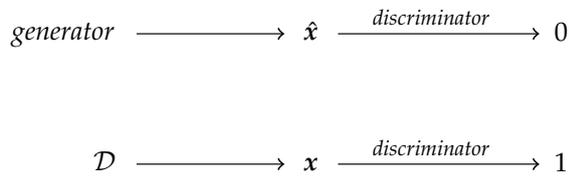
$\mathcal{D} \longrightarrow x \xrightarrow{\textit{discriminator}} 1$

Figure 2.5. Generative adversarial network architecture. The discriminator must predict 0 for data points generated by the generator, and 1 for data points from the dataset. The generator wants the discriminator to predict 1 for its generations.

3   Optical flow

Optical flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer and the scene.[3] In other words, optical flow does not represent the actual motion of the objects in the scene (that would not be computable from images), but rather the motion of the pixels in the image plane. Optical flow is merely an approximation of the motion field. But, in most cases, it is a pretty good approximation.

The problem statement of optical flow is to estimate the motion, *i.e.*, velocity, of each pixel, given two consecutive image frames. To solve this problem, optical flow makes the following two assumptions,

- *Brightness constancy*: when a point $x = [x, y]$ in the image at timepoint t moves to point $x + \delta_x$ in the image at timepoint $t + \delta_t$, its luminance does not change,

$$\mathbf{I}[x + \delta_x, y + \delta_y, t + \delta_t] = \mathbf{I}[x, y, t];$$

- *Small motion*: the pixels do not move far between image frames. The implication of this is that we can use the Taylor expansion for a very good approximation around the point we care about.

Now, we can derive how to determine the velocity $v = [u, v]$ at point $x = [x, y]$ as follows,

$$\mathbf{I}[x, y, t] = \mathbf{I}[x + \delta_x, y + \delta_y, t + \delta_t] \qquad \text{Brightness constancy}$$
$$= \mathbf{I}[x, y, t] + \mathbf{I}_x \delta_x + \mathbf{I}_y \delta_y + \mathbf{I}_t \delta_t, \qquad \text{First-order Taylor approximation}$$

where $\mathbf{I}_x, \mathbf{I}_y, \mathbf{I}_t$ are the derivatives at the implied location $[x, y, t]$. From this, we can derive the *brightness constancy equation*,

$$\mathbf{I}[x, y, t] + \mathbf{I}_x \delta_x + \mathbf{I}_y \delta_y + \mathbf{I}_t \delta_t = \mathbf{I}[x, y, t]$$
$$\mathbf{I}_x \delta_x + \mathbf{I}_y \delta_y + \mathbf{I}_t \delta_t = 0$$
$$\mathbf{I}_x \frac{\delta_x}{\delta_t} + \mathbf{I}_y \frac{\delta_y}{\delta_t} + \mathbf{I}_t = 0$$
$$\mathbf{I}_x u + \mathbf{I}_y v = -\mathbf{I}_t.$$

However, it cannot be solved, because we have two unknowns, *u* and *v*, with one equation per pixel. This is called the *aperture problem*. The way this is dealt with is the difference between the Lucas-Kanade and Horn-Schunck algorithms, *i.e.*, their difference is how they add more constraints to make the problem tractable.

3.1   Lucas-Kanade

Lucas-Kanade applies a local method that assumes that pixels in the same area, *i.e.*, a patch around the pixel, have the same velocity. This

gives us an additional constraint for every pixel in the area, which we can solve by least-squares estimation,

$$
\begin{bmatrix}
\mathbf{I}_x[x_1,y_1,t] & \mathbf{I}_y[x_1,y_1,t] \\
\vdots & \vdots \\
\mathbf{I}_x[x_n,y_n,t] & \mathbf{I}_y[x_n,y_n,t]
\end{bmatrix}
\begin{bmatrix} u \\ v \end{bmatrix} = -
\begin{bmatrix}
\mathbf{I}_t[x_1,y_1,t] \\
\vdots \\
\mathbf{I}_t[x_n,y_n,t]
\end{bmatrix}
$$

$$
A\boldsymbol{v} = -\boldsymbol{b}
$$

This is equivalent to solving the following by least-squares estimation,

$$
A^\top A \boldsymbol{v} = -A^\top \boldsymbol{b}
$$

$$
\sum_{(x,y)\in P}
\begin{bmatrix}
\mathbf{I}_x^2 & \mathbf{I}_x\mathbf{I}_y \\
\mathbf{I}_x\mathbf{I}_y & \mathbf{I}_y^2
\end{bmatrix}
\begin{bmatrix} u \\ v \end{bmatrix} = -
\sum_{(x,y)\in P}
\begin{bmatrix}
\mathbf{I}_x\mathbf{I}_t \\
\mathbf{I}_y\mathbf{I}_t
\end{bmatrix}
$$

This is the same matrix as the structure tensor we saw in the Harris corner detector. Thus, corners are good places to compute flow. If there are no corners, and *e.g.* only an edge, then we have the aperture problem.

3.2 Horn-Schunck

Horn-Schunck applies a global method that assumes optical flow fields to be smooth. It seeks to minimize the following energy function,[4]

$$
E = \int\int \underbrace{(\mathbf{I}_x u_{xy} + \mathbf{I}_y v_{xy} + \mathbf{I}_t)^2}_{\text{brightness change penalty}} + \underbrace{\lambda(\|\boldsymbol{\nabla} u_{xy}\|^2 + \|\boldsymbol{\nabla} v_{xy}\|^2)}_{\text{flow change penalty}}\, dxdy,
$$

[4] In the discrete case that we care about, the integrals are replaced by sums.

where u_{xy} and v_{xy} are the velocities at $[x,y]$ forming a flow field. The first term minimizes the brightness change, which enforces the brightness constancy assumption. The second term minimizes the change in flow, which enforces the smooth flow field assumption. In the discrete case, this can be seen as minimizing the difference in flow between neighbors, resulting in smoother flow. A larger λ leads to a smoother flow field.

The derivatives of this function are the following,

$$
\frac{\partial E}{\partial u_{kl}} = 2(u_{kl} - \bar{u}_{kl}) + 2\lambda(\mathbf{I}_x u_{kl} + \mathbf{I}_y u_{kl} + \mathbf{I}_t)\mathbf{I}_x
$$
$$
\frac{\partial E}{\partial v_{kl}} = 2(v_{kl} - \bar{v}_{kl}) + 2\lambda(\mathbf{I}_x u_{kl} + \mathbf{I}_y u_{kl} + \mathbf{I}_t)\mathbf{I}_y.
$$

$\bar{u}_{kl}$ and $\bar{v}_{kl}$ are the average flow of the 4 neighbors: $[k-1,l], [k+1,l], [k,l-1], [k,l+1]$.

The extrema of E can be found by setting the derivatives to zero and solving for the unknowns u and v. These form a linear system that can be solved iteratively using update equations,

$$
\hat{u}_{kl} = \bar{u}_{kl} - \frac{\mathbf{I}_x \bar{u}_{kl} + \mathbf{I}_y \bar{v}_{kl} + \mathbf{I}_t}{\lambda^{-1} + \mathbf{I}_x^2 + \mathbf{I}_y^2}\mathbf{I}_x
$$
$$
\hat{v}_{kl} = \bar{v}_{kl} - \frac{\mathbf{I}_x \bar{u}_{kl} + \mathbf{I}_y \bar{v}_{kl} + \mathbf{I}_t}{\lambda^{-1} + \mathbf{I}_x^2 + \mathbf{I}_y^2}\mathbf{I}_y.
$$

These update equations are used to update the flow field iteratively until convergence, starting from zero flow field.

This can also be used to compute flow field of frames with larger motions by using coarse-to-fine flow estimation. This is done by first downscaling the image until the displacement is 1 pixel. Then, compute flow of that and upscale the image and flow field. Repeat this step until you are back at the original image size.

4   Recognition

In *recognition*, we want to be able to recognize whatever (parts of) an image depicts. For example, determining where all the humans are in an image. The challenges of this task are the following,

- Background clutter;[5]

- Intra-class variations.[6]

At first (~1960–1990), the geometric era saw recognition as an alignment problem, where everything consists of shape primitives. Then, you only had to recognize the primitives and figure out their alignment to figure out what object is depicted. Later (~1990–2000), appearance-based models estimate the appearance by combining a lot of images that depict the same thing, *e.g.*, human faces. Then, new data points are compared to this estimate of appearance. Sliding window approaches were also used, which compare every patch of an image to a template. Currently, image classifiers are implemented as machine learning problems that take an image as input and outputs one of a set of predefined labels.

4.1   Bag-of-words

An example of a machine learning algorithm for classification is *bag-of-words*. At a high level, it extracts local features from an image, which are compared to the local features of the training images in an efficient and robust way. Each matching patch is weighted according to the TF-IDF metric,

$$\text{TFIDF}(t, d) = \underbrace{\frac{N_{t,d}}{N_d}}_{\text{TF}} \cdot \underbrace{\log\left(\frac{|\mathcal{D}|}{|\{d' \in \mathcal{D} \mid t \in d'\}|}\right)}_{\text{IDF}},$$

and used to determine a classification of the input image.

1. *Feature extraction*: The goal of this step is to find patches that would be interesting for the classification of its image. For this, algorithms such as SIFT and Harris detection are used to find such interesting patches. These patches are then extracted and used in the following step;

2. *Dictionary learning*: Patches that depict the same usually do not contain the exact same pixels. Thus, we need some way of mapping patches that depict the same together. This is done by k-means clustering, which iteratively updates its k clusters by assigning each object to the cluster with the nearest centroid and updating their representation (centroid) to be the mean of its objects. This step is repeated until convergence. Thus, now each feature is mapped to its closest centroid, which means that similar features are mapped together;

3. *Feature encoding*: The images are encoded by the bag-of-words vectors of their features, *i.e.*, an histogram of the counts of the number of feature occurrences;

k-means clustering is a method of partitioning observations into k clusters. It does so by iteratively alternating between updating the clusters and assigning points to clusters. It represents the clusters by their mean, and it assigns points by assigning it to the cluster representation with the smallest distance. It repeats this until convergence, which is when none of the points update their cluster assignment.

4. *Classification*: The classification of an image is the class that maximizes the TF-IDF metric,[7] which weights each feature by its inverse document frequency. *I.e.* if a feature occurs in a lot of images, it has a lower weight, since it is less discriminative.

[7] Note that each feature maps to the class of which its image is part.

5   Segmentation

Segmentation involves the grouping of pixels, where the groupings indicate a semantic relationship. For example, segmenting pixels into background and foreground. The purpose of segmentation is that the found groupings can be used in further algorithms.

5.1   k-means

The most simple way of performing segmentation is using the k-means algorithm. First, we encode all pixels into a feature space, after which we apply k-means to the features. Then, if two pixels are part of the same cluster, they are part of the same grouping in the segmentation.

Examples of features are RGB values or filter bank responses.[8] However, these do not give us spatial coherence, because spatial information is not taken into account. If there is a lot of noise, the groupings may not be very good. Thus, we want pixels that are spatially close to have a greater chance of being in the same grouping. The way to do this with k-means is to add two dimensions to the features, x and y position. This enforces spatial coherence.

[8] Apply 24 common filters by convolution.

The advantage of k-means is that it is simple and fast. However, it has many problems,

- There is no way of knowing what the value of k should be;

- k-means is sensitive to initial centers;

- k-means is sensitive to outliers;

- It only detects spherical clusters, because points are assigned to clusters by their squared distance.

5.2   Mixture of Gaussians

Mixture of Gaussians (MoG) solves the problem of k-means' sensitivity to outliers. Instead of treating the pixels as a bunch of points, we will assume that they are all generated by sampling a continuous function. Then, we will be able to remove outliers, because their likelihood will be very low.

MoGs assume that the data is generated by k weighted d-dimensional Gaussians with means $\boldsymbol{\mu}_b \in \mathbb{R}^d$, covariance matrices $\boldsymbol{\Sigma}_b \in \mathbb{R}^{d \times d}$, and weights α_b. The likelihood of observing $\boldsymbol{x}$ is the following,

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \sum_{b=1}^{k} \alpha_b \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_b, \boldsymbol{\Sigma}_b).$$

We then want to optimize the model w.r.t.

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\mu}_1 & \cdots & \boldsymbol{\mu}_k & \boldsymbol{\Sigma}_1 & \cdots & \boldsymbol{\Sigma}_k & \alpha_1 & \cdots & \alpha_k \end{bmatrix},$$

which is done with the *expectation-maximization* (EM) algorithm. EM works by estimating the "ownership" each Gaussian has over the points, given the current estimate of θ (E-step). Then, we update θ given the "ownership" (M-step). This is done iteratively.

In more details, given the current estimate θ, the E-step computes the probability that point x is in blob b, which we call the "ownership" of b over x,

$$p(b \mid x, \mu_b, \Sigma_b) = \frac{\alpha_b p(x \mid \mu_b, \Sigma_b)}{\sum_{b'=1}^{k} \alpha_{b'} p(x \mid \mu_{b'}, \Sigma_{b'})}.$$

Given the "ownership" information, the M-step computes the new means, covariance matrices, and weights,

$$\alpha_b' = \frac{1}{n} \sum_{i=1}^{n} p(b \mid x_i, \mu_b, \Sigma_b)$$

$$w_i^{(b)} = \frac{p(b \mid x_i, \mu_b, \Sigma_b)}{\sum_{j=1}^{n} p(b \mid x_j, \mu_b, \Sigma_b)}$$

$$\mu_b' = \sum_{i=1}^{n} w_i^{(b)} x_i \qquad \text{Weighted mean}$$

$$\Sigma_b' = \sum_{i=1}^{n} w_i^{(b)} (x_i - \mu_b)(x_i - \mu_b)^\top. \qquad \text{Weighted variance}$$

Then, after running the EM algorithm, we assign each pixel to the blob with the highest "ownership" over the pixel.

The advantage of this approach is that it is probabilistic and can predict new data points, because it is generative. This also has the advantage that we can detect outliers. Furthermore, it can be stored in $\mathcal{O}(kd^2)$, which is quite compact. The problems are that we still need to know the number of components k, and we need a good initialization.[9]

[9] It is often a good idea to start from the output of k-means.

5.3   Mean-shift

Mean-shift is an algorithm that does not require a predetermined amount of clusters, like k-means and MoG. Instead, it localizes the histogram modes of the data. Then, all identified modes are considered a cluster. It works by starting from random points and iteratively moving toward the center of mass of the window around the point. This will cause the algorithm to move toward high density areas. Ultimately, all points that move toward the same area, *i.e.* same mode, are considered a cluster.

The iterative update rule is the following,

$$f(x) = \frac{1}{\sum_{i=1}^{n} k(x, x_i)} \sum_{i=1}^{n} x_i k(x, x_i),$$

where x is the current point, k is a distance function, and $f(x)$ is the new point. Intuitively, it is a weighted average of its distance to all other

points, which makes it move toward high density areas. The attraction basin is the region for which all trajectories lead to the same mode, and all data points in the same attraction basin of a mode are considered a cluster.

The advantage of this method is that it does not require a predetermined amount of clusters. Another advantage is that it does not assume any prior shape on data clusters. Furthermore, it has just a single parameter h, which is the window size, so it has a physical meaning. However, the selection of h is not trivial, and the method is computationally quite expensive.

5.4 Hough transform

Remark. This is not an algorithm for segmentation, but rather for finding features.

The *Hough transform* uses the structure of shapes to recognize them in an image. First, we have to find the edges of the objects in the image, so we can fit shapes to them. Now, let's say that we want to detect straight lines for example.[10] We use the following equation for a line,[11]

$$x \cos \theta + y \sin \theta = \rho.$$

Then, we represent all lines through all the points on the edges in a parameter space. *E.g.*, the point $[0, 1]$ lies on all lines with $\rho = 1$, thus all lines in this point are represented by the line $\rho = 1$ in parameter space. Then, we need to identify peaks in the parameter space. These peaks make up the geometry of the objects in the image.

The advantage of this approach is that it can be done for any shape, such as a circle, but the problem is that as the parameters grow, it gets exponentially more expensive.

5.5 Interactive segmentation

In *interactive segmentation*, we assume that we also have information from the user about the segmentation. *I.e.*, the user has identified some pixels as being part of some grouping. We can use this information to provide a better segmentation.

In this approach, we will assume that the pixels of the image form a *Markov random field* (MRF). We assume that there is a defined relationship $\psi(x_i, x_j)$ between adjacent pixels and a defined relationship $\phi(x_i, y_i)$ between pixels and their hidden state (label). The joint probability of the field is then defined as the following,
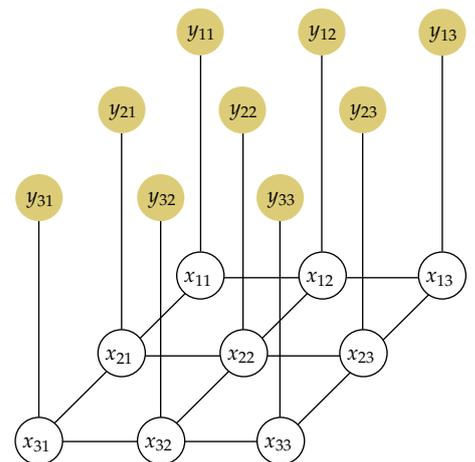
$$p(X, y) = \prod_i \phi(x_i, y_i) \prod_{i,j} \psi(x_i, x_j).$$
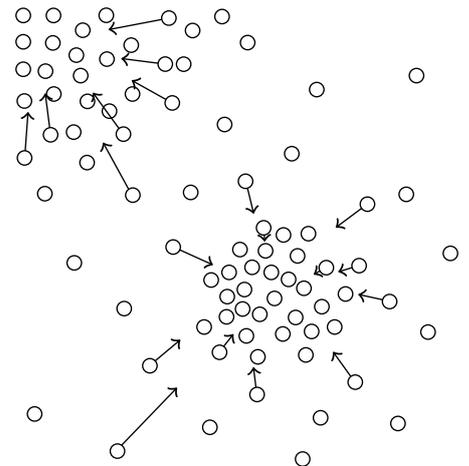
Figure 5.1. Iteration of mean-shift where points move toward a mode.

[10] These are important, because of their omnipresence in man-made scenes.
[11] We use this instead of $y = ax + b$, because this has a bounded parameter domain.

Figure 5.2. Markov random field, where the y values are observed, while the underlying x values remain unobserved.

We then want to minimize the negative log-likelihood,

$$E(\boldsymbol{X}, \boldsymbol{y}) = -\sum_i \phi(\boldsymbol{x}_i, y_i) - \sum_{i,j} \psi(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

We call this the energy function. We can then use an inference algorithm such as Gibbs sampling to minimize the energy function. However, the most popular one is GraphCut.

The idea of *GraphCut* is to turn the MRF into a source-sink graph, where the edge weights are defined by ϕ and ψ. Let's say we want to segment the image into foreground/background[12] and we have input from the user about some pixels that are background and some that are foreground. We now want to know how to assign the surrounding pixels. First, we convert all the known background pixels into sources and foreground pixels into sinks in the source-sink graph. Then, we find the minimum cost cut that separates the background from the foreground and label the pixels appropriately. This enforces spatial coherence.

We can also minimize the energy function further by iteratively alternating between using GraphCut and fitting an MoG. Starting from the user-provided bound, each iteration, we get a new bound from Graph-Cut, which is used to refine the MoG, which is used to refine the bounds with GraphCut, etc.

5.6   Learning-based approaches

In fully supervised learning, we assume we have access to a training dataset of images with full segmentation mask as labels. Thus, whatever type of segmentation that will be done by a model will depend on the training data. If the training data consists of background/foreground segmentation, the model will do background/foreground segmentation.

Models are evaluated with the intersection over union (IoU) metric (see Figure 5.3),

$$\frac{1}{C} \sum_{i=1}^{C} \frac{n_{ii}}{n_i + \sum_{j=1}^{C} n_{ji} - n_{ii}},$$

where C is the amount of labels, n_i is the amount of pixels of label i, and n_{ij} is the amount of pixels of class i predicted to have class j.

Like in unsupervised approaches, we first need to extract features of the pixels, which can be learned. These features are then given as input to the models, which are optimized to predict the training labels.

k-nearest neighbors.   The simplest learning-based approach is *k*-nearest neighbors (kNN). In kNN, every point is labeled to be the mode of its *k* nearest neighbors in the training data. The "nearest" neighbors refer to the points in the feature space with the smallest distance to the current feature vector, not the nearest point in the image.

$$\text{IoU} = \frac{A \cap B}{A \cup B}$$
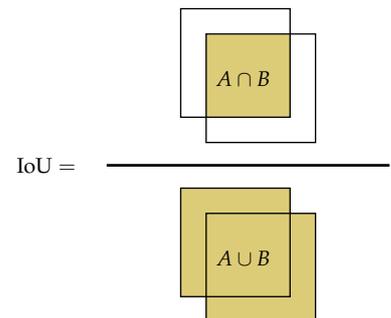
Figure 5.3. Illustration of the intersection over union metric.

Transfer learning.   Transfer learning is a method that does not require a large dataset.[13] In transfer learning, the weights are first learned on a related task, such as classification, and dataset (pretraining). Then, we initialize our segmentation model with these weights as much as possible and train on the actual dataset (finetuning). This makes the assumption that pretrained weights are easier to train for segmentation than a random initialization.

Hypercolumns.   It is common to pretrain on a classification task (one label per image) and then augmenting the model for segmentation (one label per pixel). Hypercolumns do this by taking the pixel activations for each pixel at each layer of the classification network, concatenating them, and calling that the pixel's embedding (see Figure 5.4).[14] This embedding is then used as input to a pixel classification network that actually predicts the grouping of the pixel.

This implicitly enforces spatial coherence, because convolutional layers work locally. Thus, close activations are closely related. The results of hypercolumns are pretty good, but the boundaries are very coarse, due to the downsampling layers of the classification model, which cause many features to be the same value for close pixels.

Fully convolutional networks.   In fully convolutional networks (FCN), we change the usually used linear layers of the classification network to 1×1 convolutional layers. This allows for pretraining on a classification task, because linear layers and 1×1 convolutional layers have the same parameters. Then, at the end of the model, the images are not turned into vectors, but rather feature maps. Then dependent on how much the images were downsampled by the model, we upsample the feature map from different layers of the network to be of same size as the input image with a transposed convolution. Now, we have features per pixel that can be used to predict the label per pixel with 1×1 convolution.

Refinement with conditional random fields.   The outputs of FCNs are not sharp. But, we can use an additional refinement step with a conditional random field (generalization of Markov random field). We can use the FCN output as the unary term ϕ and an edge-aware pairwise term ψ to refine the segmentation.

U-nets.   This model is similar to FCNs, but rather than doing the upsampling in a single transposed convolutional layer, we use the same amount of upsampling layers as downsampling layers. Additionally, U-nets also use skip-connections between downsampling and upsampling layers of the same level.

Dilated convolutions.   In computer vision, we need convolutional layers to process an input image for a task, and downsampling layers to make the
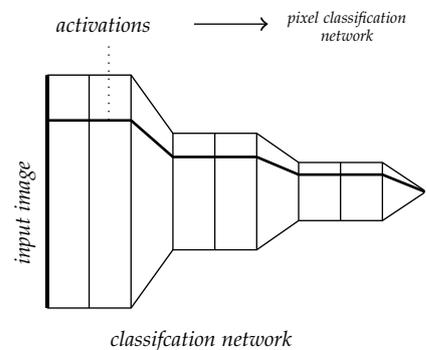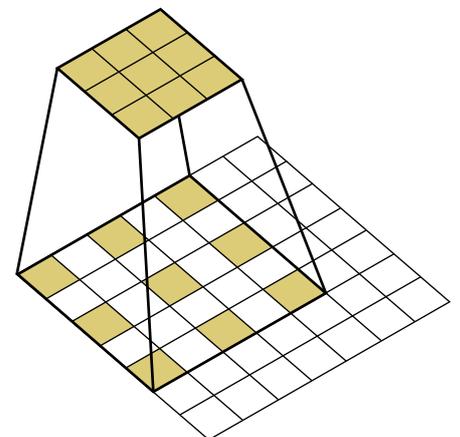
Figure 5.4. Hypercolumns.

Figure 5.5. Dilated convolution

receptive field large. However, because the convolutional layers work very locally (usually 3×3 or 5×5), we miss out on contextual information, and because the downsampling layers remove data, we lose information. We want a layer that does not downsample, but still has a large receptive field. The dilated convolution is a generalization of the convolutional layer that skips values for an exponentially large receptive field without a change in the amount of weights (see Figure 5.5). Thus, we can still pretrain a classification network and reuse the weights in the dilated layers. Another advantage of not needing downsampling layers is that we get an output for every pixel without needing to upsample, which is what we want in segmentation.

However, simply only using dilated layers causes artifacts, because the learned filters are often discontinuous. To mitigate this, we can add additional convolutional layers to the beginning and end of the model to smooth out the artifacts. This results in much better segmentations.

6   Object detection

In *object detection*, we want to detect objects within an environment. We are given an image as input, and the output should be a set of objects with their location within the image.

6.1   Detection via classification

The most naive idea is to run a classification model on every patch within the image. We can use all previously discussed concepts for the representation of the patches, such as a gradient histogram or pixel intensities.

6.2   Boosting with weak classifiers

An example of a weak classifier is the rectangle filter. It is essentially a large gradient calculation, where we compute the sum of one patch, minus the sum of another next to it. We can compute these very quickly using a sliding window approach, for which we only need to compute a cumulative sum over the integral image in $\mathcal{O}(HW)$. Then, we can compute the sum of any patch in $\mathcal{O}(1)$. We then classify the patch based on the difference of sums of the patches.

This will result in a single very weak classifier, but if we combine a lot of them, we can use boosting to combine them linearly to be a strong classifier. This is fast, because each weak classifier is very simple. Boosting works by iteratively finding the weak classifier that achieves the lowest weighted training error (weights are initialized to be uniform). Then, we raise the weights of training samples that were misclassified by the best weak classifier. We repeat this, while collecting the best weak classifier every iteration to get a subset of weak classifier that collectively are strong. We weight each weak classifier by their accuracy on the training dataset.

This will result in a subset of weak classifiers that each might have a little better than expected accuracy, but together they form a strong classifier that has good accuracy.

In more details, AdaBoost [Freund et al., 1996] runs for M iterations, each resulting in a weak classifier. Each iteration m, it first trains a new weak classifier h_m that minimizes the weighted error function,

$$J_m = \sum_{i=1}^{n} w_i^{(m)} \mathbb{1}\{h_m(x_i) \neq y_i\}.$$

Then, it estimates the weighted error of this classifier on the dataset,

$$\epsilon_m = \frac{1}{\sum_{i=1}^{n} w_i^{(m)}} \sum_{i=1}^{n} w_i^{(m)} \mathbb{1}\{h_m(x_i) \neq y_i\}.$$

Then, it calculates the weighting coefficient for $h_m(x)$, which is large if

the weak classifier is good, and otherwise small,

$$\alpha_m = \log\left(\frac{1 - \epsilon_m}{\epsilon_m}\right).$$

Lastly, it updates the weighting coefficients,

$$w_i^{(m+1)} = w_i^{(m)} \exp(\alpha_m \mathbb{1}\{h_m(x_i) \neq t_i\}).$$

The final classifier is then the weighted linear combination, according to α_m, of the weak classifiers.

6.3 *Implicit shape model*

An implicit shape model (ISM) learns a star-topology structural model, which means that features are considered independent given the center of the object. It learns a visual codebook with displacement vectors that encode where the center of the object must be that this visual word is a part of. For example, we might encode that a car must have two visible wheels. Then, if two wheels point at the same center with their displacement vectors, there is a car object at that center.

6.4 *Learning-based approaches*

The difficulty in using learning-based approaches in object recognition is that the output has a variable size, since an image may contain any number of objects. Thus, we must design new models that can output a variable amount of outputs. Also, there are an exponential amount of possible regions within an image, which makes classifying every possible patch infeasible.

The solution to this is selective search [Uijlings et al., 2013]. Selective search first segments the image into a large number of regions using a graph-based segmentation technique. Then, it iteratively alternates between collecting segmentation regions as region proposals, and combining segmentation regions into larger ones. This results in a wide spectrum of region proposal sizes. This enables learning-based approaches, because now we only need to classify a small number of patches.

R-CNN. R-CNN works by first extracting approximately 2000 region proposals using selective search [Girshick et al., 2014]. Then, it warps the region proposals to constant-sized images patches and passes them to a classification network, which classifies the region (CNN followed by support vector machine).

The problem with R-CNN is that the training and testing is still slow (47 seconds per test image).

Fast R-CNN. Fast R-CNN solves the problem of R-CNN by first feeding the entire input image into a CNN to generate a feature map [Girshick,
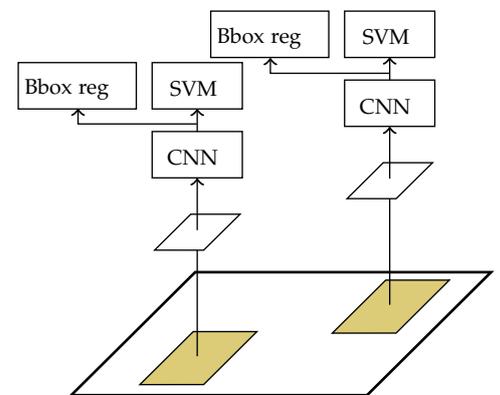

Figure 6.1. R-CNN architecture.

2015]. Then, the region proposals are identified within this feature map. This is then reshaped to a constant-sized representation which is passed to a fully connected layer that outputs a probability distribution over classifications. The reason for that this model is faster is because we only need to run a single CNN, instead of 2000 for each region proposal.

Faster R-CNN.   The largest bottleneck in Fast R-CNN is the selective search, which is very slow. Therefore Ren et al. [2015] came up with Faster R-CNN that uses a region proposal network (RPN) that lets the model learn the region proposals. The RPN works by taking the feature map as input and outputting a bounding box and "objectness" score.

This network has four loss functions that need to be optimized: Region proposal "objectness" classification loss, Region proposal bounding box regression loss, Region classification loss, and another bounding box regression loss on the eventual classification network.

You only look once.   All R-CNN approaches require a call to a classification network for every region proposal. However, this slows down the algorithm. YOLO predicts bounding boxes and class probabilities in a single pass [Redmon et al., 2016]. It works by dividing the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell then predicts B bounding boxes ((x, y, w, h, c), where c is the confidence score), and C conditional class probabilities.[15] It then picks the boxes with the greatest confidence and removes boxes if their IoU is lower than some threshold.

YOLO is much faster than the R-CNN approaches with a real-time performance of approximately 45 frames per second. A minor problem is that it struggles with small objects and it imposes strong spatial constraints (since each grid cell can only have one class prediction), which limits the number of nearby objects it can predict.
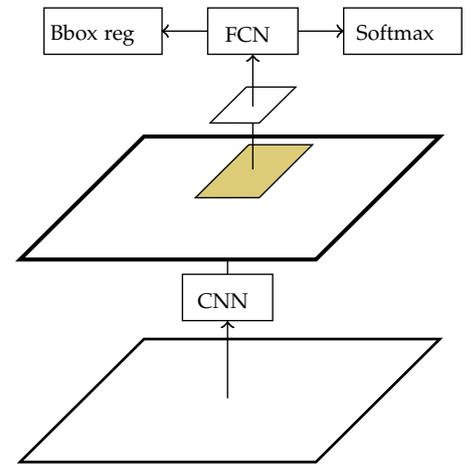

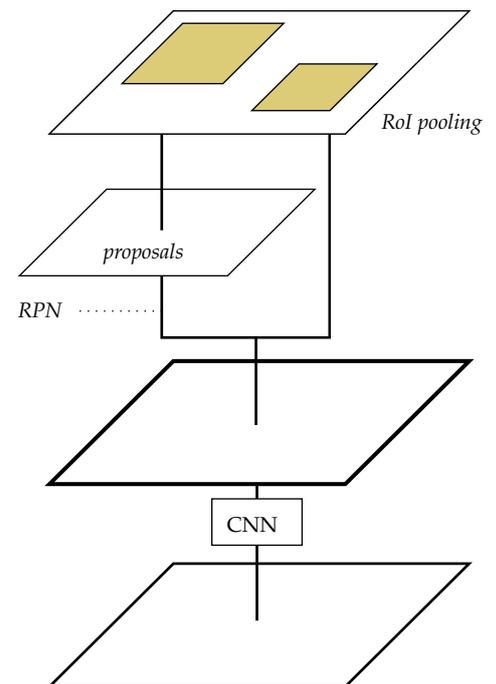**Figure 6.2.** Fast R-CNN architecture.

Figure 6.3. Faster R-CNN architecture.

[15] Thus, the output of the model has dimensionality $S \times S \times (5B + C)$.

7   Tracking

Tracking is the following of the movement of an object (point, region, or template), *i.e.*, we want to find the position of the object in the consecutive frames. Thus, the problem statement is that we want to find the position of an object in frame $t + 1$, given that we know its position in frame t.

To make sure that the object detection location does not "teleport", we can give the models the prior that objects have a constant velocity.[16] We can do this by penalizing points that are not in the direction of the current velocity of the object.

7.1   Point

If we want to track a point between frames, we can make the assumption that the color of the point will remain the same, and optimize the following error function (one-dimensional case),

$$E(h) = (\mathbf{I}[x,t]) - \mathbf{I}[x+h, t+1])^2$$
$$\approx (\mathbf{I}[x,t] - \mathbf{I}[x, t+1] - h\mathbf{I}_x[x,t])^2$$

First-order Taylor approximation

$$\frac{\partial}{\partial h} E(h) = -2\mathbf{I}_x[x,t](\mathbf{I}[x,t] - \mathbf{I}[x, t+1] - h\mathbf{I}_x[x,t]).$$

At the minimum, where the derivative is 0, we get the following,

$$h = \frac{\mathbf{I}[x, t+1] - \mathbf{I}[x,t]}{\mathbf{I}_x[x,t]}.$$

However, if the gradient is 0, *i.e.*, when the image region is flat, $\mathbf{I}_x[x,t]$ will be 0 and thus we have no information about which direction the point moved in. This is very similar to the problem in optical flow, called the aperture problem.

Furthermore, we assume to always move toward the closest minimum. But, the image is non-convex in general, which has multiple solutions. This can be solved by a high framerate, because then the movement between frames is small.

Like in optical flow, in two dimensions, we need more constraints to solve for the displacement. And, like in optical flow, we can solve this by either solving it globally (Horn-Schunck) or locally (Lucas-Kanade).

7.2   Template

In template tracking, we want to track an object between frames, represented by a bounding box, which we call its template. The most naive solution is to simply compare pixel intensities and minimizing the following error function,

$$E(u, v) = \sum_{x,y} (\mathbf{I}[x+u, y+v] - \mathbf{T}[x,y])^2,$$

where $\mathbf{T}$ is the template. However, the object in the template might make a transformation such as a rotation or scaling. We could also parametrize those transformations, instead of only a translation, by generalizing the previous error function to any transformation family W,

$$E(\boldsymbol{p}) = \sum_x (\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] - \mathbf{T}[\boldsymbol{x}])^2,$$

where $\boldsymbol{p}$ parametrizes a "warp" W. However, we cannot directly solve for $\boldsymbol{p}$, thus we iteratively improve it by updating it with some $\delta_{\boldsymbol{p}}$,

$$= \sum_x \big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p} + \delta_{\boldsymbol{p}})] - \mathbf{T}[\boldsymbol{x}]\big)^2$$

$$\approx \sum_x \big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_{\boldsymbol{p}} \nabla_{\boldsymbol{p}} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] - \mathbf{T}[\boldsymbol{x}]\big)^2 \qquad \text{First-order Taylor approximation}$$

$$= \sum_x \big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_{\boldsymbol{p}} \nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) - \mathbf{T}[\boldsymbol{x}]\big)^2 \qquad \text{Chain rule}$$

$$\frac{\partial}{\partial \delta_{\boldsymbol{p}}} E(\boldsymbol{p}) = 2 \sum_x \Big(\nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) \Big)^{\top} \qquad \text{Chain rule}$$
$$\Big(\mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] + \delta_{\boldsymbol{p}} \nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) - \mathbf{T}[\boldsymbol{x}] \Big).$$

We can then solve for $\delta_{\boldsymbol{p}}$ by setting the gradient to 0,

$$\delta_{\boldsymbol{p}} = \boldsymbol{H}^{-1} \sum_x \Big(\nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) \Big)^{\top} (\mathbf{T}[\boldsymbol{x}] - \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})]),$$

with Hessian matrix $\boldsymbol{H}$ defined as follows,

$$\boldsymbol{H} = \sum_x \Big(\nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) \Big)^{\top} \Big(\nabla_{W(\boldsymbol{x};\boldsymbol{p})} \mathbf{I}[W(\boldsymbol{x}; \boldsymbol{p})] \nabla_{\boldsymbol{p}} W(\boldsymbol{x}; \boldsymbol{p}) \Big).$$

We iteratively update the warp by $\delta_{\boldsymbol{p}}$ until convergence.

However, this method is not robust to image noise, since it assumes that pixel intensities remain the same between frames. Furthermore, some three-dimensional transformations are impossible to parametrize as a two-dimensional warp of an image. For example, a person turning around w.r.t. the camera cannot be parametrized as a two-dimensional warp.

7.3 Tracking by detection

We can also track by detecting keypoints in each frame and minimizing the distance to the feature descriptor of the point we are tracking. This does not require the assumption that the image intensities remain the same, and can track large displacements.

We can also use this to track a region by matching keypoint descriptors of the template with the next frame. Then, removing outliers with *e.g.* RANSAC. The bounding box in the next frame is then the box that encapsulates all points that are left.

However, in many cases, we do not have an exact template, but we want to track any object of a specific type. This can be done by detecting

the objects independently in each frame and then associating the detections over time with a bipartite matching algorithm. This is especially important when there are multiple objects in the scene.

7.4 Online learning

Often, the template changes throughout frames, but, until now we have assumed that the template is constant. However, it is more realistic to assume that the appearance of the tracked object changes. Online learning accounts for this by collecting all detected features of the tracked object throughout the frames and the background. Then, it uses these to train the detector every frame. Thus, the detector improves every frame, and updates it representation of the object.

The advantage of this is that it is robust to changes to the environment, *e.g.*, if we go from a dark to a bright environment. However, the disadvantage is that the representation of the object can gradually drift to something else. But, this can be avoided by not allowing the model to drift too far from our initial template with additional constraints.

8   Projective geometry

Using *projective geometry*, we can model the imaging process of a perspective camera,[17] because it satisfies the following constraints,

- Straight lines must be mapped to straight lines;

- The size of an object must be inversely proportional to its distance from the camera;

- We must be able to represent points at infinity to model vanishing points.

In projective geometry, the homogeneous coordinate space is used,

$$\tilde{x} \propto \begin{bmatrix} x \\ 1 \end{bmatrix} \in \mathbb{P}^2,$$

which can be converted to Euclidean space,

$$\tilde{x} \mapsto \begin{bmatrix} x/w \\ y/w \end{bmatrix} \in \mathbb{R}^2,$$

where w is the last entry of $\tilde{x}$. Homogeneous coordinates with $w = 0$ are called *ideal points* and serve as a direction toward infinity, rather than points.

In this space, lines can be represented as the following,

$$\ell = \begin{bmatrix} a & b & c \end{bmatrix}^\top,$$

where the points on this line satisfy $\ell^\top \tilde{x} = 0$. Now, if we want to find the point where two lines ℓ, ℓ' intersect, we need to find a point that satisfies $\ell^\top \tilde{x} = \ell'^\top \tilde{x} = 0$. This is exactly the cross product,

$$\tilde{x}_{\text{intersection}} = \ell \times \ell'.$$

We can find the line between two points $\tilde{x}_1, \tilde{x}_2$ in the same way,

$$\ell = \tilde{x}_1 \times \tilde{x}_2.$$

In three dimensions, we can also represent planes,

$$\pi = \begin{bmatrix} a & b & c & d \end{bmatrix}^\top.$$

A point lies on the plane π if and only if $\pi^\top X = 0$. If we want to find a plane from three points, we need to solve

$$\pi^\top \begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} = 0.$$

We can construct lines as the intersection of two planes, or construct points as the intersection of three planes.

8.1   Transformations

Translations in Euclidean geometry are non-linear, but by using homogeneous coordinates it becomes a linear operation,

$$
\begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ w \end{bmatrix}.
$$

Furthermore, we can make the following transformations,

- Rigid transformations (rotation and translation) with the following matrix,

$$
\begin{bmatrix} \cos \theta & -\sin \theta & t_x \\ \sin \theta & \cos \theta & t_y \\ 0 & 0 & 1 \end{bmatrix}.
$$

This matrix has 3 degrees of freedom: t_x, t_y, θ.

- Similarity transformations (rotation, translation, and scaling) with the following matrix,

$$
\begin{bmatrix} s \cos \theta & -s \sin \theta & t_x \\ s \sin \theta & s \cos \theta & t_y \\ 0 & 0 & 1 \end{bmatrix}.
$$

This matrix has 4 degrees of freedom: t_x, t_y, θ, s.

- Affine transformations (rotation, translation, scaling, and shearing) with the following matrix,

$$
\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix}.
$$

This matrix has 6 degrees of freedom: $a_{1:6}$

- Projective transformations with the following matrix,

$$
\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.
$$

This matrix has 8 degrees of freedom, because it is specified up to scale.

8.2   Homography

Suppose that we have point correspondences between two images. Now, we want to find the projective transformation H that relates the two images. For this, we use *direct linear transformation*. We have the following equations,

$$
\lambda_i \bar{x}'_i = H \bar{x}_i
$$

The homography is constrained to map x_i to x'_i.

$$
\begin{bmatrix} \lambda_i x'_i \\ \lambda_i y'_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} h_{11} x_i + h_{12} y_i + h_{13} \\ h_{21} x_i + h_{22} y_i + h_{23} \\ h_{31} x_i + h_{32} y_i + h_{33} \end{bmatrix}.
$$

Now, when we revert them back to the real coordinate system, we get the following,

$$\lambda_i = h_{31}x_i + h_{32}y_i + h_{33}$$
$$\lambda_i x'_i = h_{11}x_i + h_{12}y_i + h_{13}$$
$$\lambda_i y'_i = h_{21}x_i + h_{22}y_i + h_{23}$$

This gives us the following linear constraints,

$$h_{21}x_i + h_{22}y_i + h_{23} - h_{31}y'_i x_i - h_{32}y'_i y_i - h_{33}y'_i = 0$$
$$h_{11}x_i + h_{12}y_i + h_{13} - h_{31}x'_i x_i - h_{32}x'_i y_i - h_{33}x'_i = 0.$$

In matrix form, we can set this up as the following for n points,

$$
\begin{bmatrix}
x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 & -x'_1 \\
0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1 x_1 & -y'_1 y_1 & -y'_1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_n & y_n & 1 & 0 & 0 & 0 & -x'_n x_n & -x'_n y_n & -x'_n \\
0 & 0 & 0 & x_n & y_n & 1 & -y'_n x_n & -y'_n y_n & -y'_n
\end{bmatrix}
\begin{bmatrix}
h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33}
\end{bmatrix} = \mathbf{0}.
$$

If we stack $n \geq 4$ points, we can use SVD to find the $\mathbf{h}$ vector that minimizes the algebraic distance. However, this does not have a geometric meaning. We would like a cost function that minimizes the distortion of the points. An example of this is the symmetric transfer error,

$$\hat{H} = \underset{H}{\operatorname{argmin}} \sum_{(x,x') \in \mathcal{D}} d(\bar{x}, H^{-1}\bar{x}')^2 + d(\bar{x}', H\bar{x})^2,$$

where $\mathcal{D}$ is the collection of point correspondences, and $d(\cdot, \cdot)$ is a distance function in real space. In practice, we use the algebraic solution as a starting point and then refine it to minimize the geometric error.

Singular value decomposition (SVD) decomposes a matrix $A = U\Sigma V^\top$, where U and V are unitary matrices, and Σ is a diagonal scaling matrix. The SVD is very useful for solving (overdetermined) equations of the following form,

$$A x = \mathbf{0}.$$

Now, when solving such equations, we want to minimize $\|Ax\|$ subject to $\|x\| = 1$. We can solve for this by using SVD,

$\min \|Ax\|$	$\|x\| = 1$
$\min \|U\Sigma V^\top x\|$	$\|x\| = 1$
$\min \|\Sigma V^\top x\|$	$\|x\| = 1$
$\min \|\Sigma b\|$	$\|Vb\| = 1.$

Then, $\|\Sigma b\|$ is minimized if

$$b = \begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix}^\top.$$

Thus, the solution is the following,

$$x = V \begin{bmatrix} 0 & \cdots & 0 & 1 \end{bmatrix}^\top = V_n,$$

where V_n is the last column vector of V.

9   Camera model

The pinhole camera is a mathematical model of how perspective cameras work. It assumes that the pinhole is infinitely small, so it only lets one lightray through per point in the space.[18]

9.1   Internal camera matrix

Figure 9.1. Illustration of how the intrinsic camera matrix works.
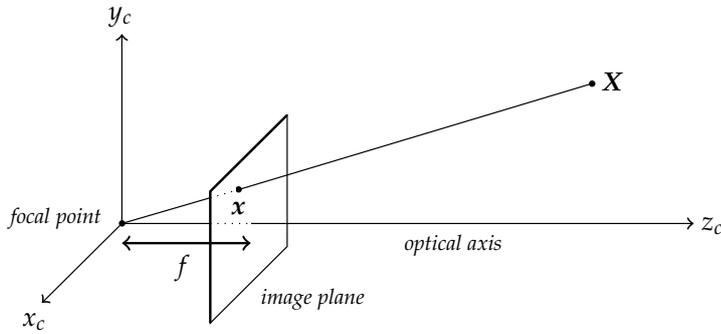
The coordinate frame of the pinhole camera is chosen such that the origin is in the center of the pinhole. The image plane behind the camera is in the plane where $z_c = -f$, where f is the focal distance. The z-axis is called the principal axis and goes in the direction where the camera is pointing toward. If we know a 3-dimensional point X_c, then it is possible to calculate the image coordinate x of that point projected onto the image plane:

$$x = -f\frac{x_c}{z_c}, \quad y = -f\frac{y_c}{z_c}.$$

The minus sign indicates that the projected image is upside down. To get rid of this mirroring, we use a virtual pinhole camera in which the image plane is put at $z_c = f$, *i.e.*, in front of the pinhole. In this model, we have

$$x = f\frac{x_c}{z_c}, \quad y = f\frac{y_c}{z_c}.$$

Using homogeneous coordinates, we can model this with the following,

$$\tilde{x} = \begin{bmatrix} f_x & s & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{X}_c = K\tilde{X}_c.$$

In practice, we often set $f_x = f_y$ and $s = 0$. The translation c moves the origin of the image from the middle to the top left of the image plane, which makes computation easier.[19]

9.2   External camera matrix

The camera will generally not be in the origin of the 3-dimensional world space. So, we need the transformation that moved the camera from the

origin to its point. Let

$$F = \begin{bmatrix} R & t \\ 0^\top & 1 \end{bmatrix}$$

be this transformation, then the projection of a point X_w in world space onto the image plane is given by the following,

$$\tilde{x} = KF^{-1}\overline{X}_w.$$

We take the inverse of F, because we want to move the origin of the points to the origin of the camera.

We denote this matrix by $P = KF^{-1}$,[20] and can be estimated by DLT with 6 (X_w, x) correspondences. Again, DLT only minimizes the algebraic error, which we can then perfect by minimizing a geometric error.

[20] P has 11 degrees of freedom.

10   Epipolar geometry

Remark. In this section, all points are in projective space. They are not denoted by $\tilde{x}$ and $\tilde{X}$, but rather by x and X.

Epipolar geometry is the geometry of stereo vision, where two cameras view a scene. When two cameras view a scene from distinct positions, there are geometric relations between the 3-dimensional points and their 2-dimensional projections on the image planes. These geometric relations lead to constraints between the image points that we can make use of. These relations are derived based on the assumption that the two cameras are approximated by the pinhole camera model (section 9).
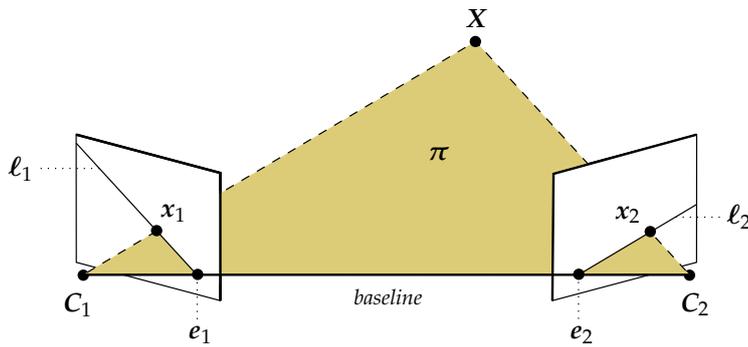
Figure 10.1. Epipolar geometry. The points x_1, x_2 on the two image planes that correspond to the same 3-dimensional point X are connected by a common plane. The intersection of the plane with the image planes form the epipolar lines.

Figure 10.1 shows the geometric relationships between the 3-dimensional points and its projections onto the images. X is the 3-dimensional point that is projected onto camera 1 as x_1 and camera 2 as x_2 (as seen in the pinhole camera model). Together, C_1, C_2, and X form the *epipolar plane π*. x_1 and x_2 must also be on this plane. More specifically, the intersections between the epipolar plane and image planes are lines called the *epipolar lines ℓ_1 and ℓ_2*. x_1 must be on ℓ_1 and x_2 must be on ℓ_2.

We can also see this from a different perspective. The projections of C_2 onto image 1, and C_1 onto image 2, are called the *epipoles e_1 and e_2*. These are computed as follows,

$$e_1 = P_1 C_2$$
$$e_2 = P_2 C_1,$$

where P_1 and P_2 are the camera matrices of cameras 1 and 2, respectively. The line formed between x_1 and e_1 is the epipolar line (same for x_2 and e_2). This gives us the following constraint,

$$\ell_1 = e_1 \times x_1$$
$$\ell_2 = e_2 \times x_2.$$

Intuitively, this is because both C_1 and C_2 are on the epipolar plane, thus so must be e_1 and e_2. The epipolar lines are the intersection of the

epipolar plane with the image planes, and we know two points that are on both, x and e, which form a line.

Notice that x_1 cannot be "behind" e_1, because that would imply that X is behind C_1, which is not the case, because X is visible in image 1. The same holds for camera 2.

In general, we want to be able to reconstruct 3-dimensional points from 2-dimensional point correspondences between n images. This is done in the following three steps,

1. Compute the fundamental (or essential) matrix that relates the point correspondences,
$$x_2^\top F x_1 = 0,$$
   where F transforms x_1 to the epipolar line of x_2 and $F^\top$ transforms x_2 to the epipolar line of x_1. This is done using the direct linear transform;

2. From the fundamental matrix, compute the relative pose (T, R) between the cameras;

3. Using the point correspondences and the relative pose, backproject the 2-dimensional image points into 3-dimensional space and find their intersection to find X.

10.1   Correspondence geometry

We need to find out in what way x_1 constrains the location of x_2 in image 2, without knowing X. x_1 constrains the point X to be on the projection line from C_1 to x_1, as can be seen in Figures 9.1 and 10.2.
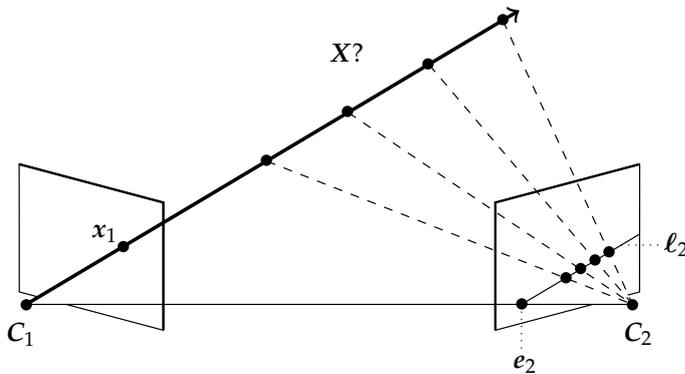
Figure 10.2. Knowing x_1 constrains its corresponding point x_2 to be on the epipolar line ℓ_2, and not behind e_2, because then X would be behind camera 1, which is not the case.

The key insight into constraining x_2 is that we do not need to know the exact location of X. We only need to find a second point on ℓ_2 to compute it with the cross product (we already know e_2). Furthermore, the projection line emanating from x_1 is on the epipolar plane, thus any point on this projection line is also on the epipolar plane. If we project that point onto image plane 2, we get a second point on ℓ_2. We can compute a point on the projection line by using the pseudo-inverse

P_1^+.[21] A 3-dimensional point X' on the projection line between C_1 and x_1 can be computed as follows,

$$X' = P_1^+ x_1.$$

The epipolar line ℓ_2 can be computed by the cross product between the epipole e_2 and this point on image plane 2,

$$\ell_2 = P_2 C_1 \times P_2 P_1^+ x_1.$$

Definition 10.1 (Fundamental matrix). The fundamental matrix F is defined as satisfying the *epipolar constraint* for all point correspondences in image 1 and 2,

$$x_2^\top F x_1 = 0.$$

The fundamental matrix relates points in one image plane with lines in the other. *I.e.*, x_2 must be on the line $F x_1$ and x_1 must be on the line $x_2^\top F = F^\top x_2$.

It satisfies the following equations w.r.t. the camera matrices,

$$F = [P_2 C_1]_\times P_2 P_1^+$$
$$F^\top = [P_1 C_2]_\times P_1 P_2^+.$$

Definition 10.2 (Essential matrix). The essential matrix E relates points in one image plane with lines in the other by the epipolar constraint in the same way as the fundamental matrix (definition 10.1),

$$\hat{x}_2^\top E \hat{x}_1 = 0,$$

where $\hat{x} = K^{-1} x$.

The difference is that the essential matrix assumes that the cameras are calibrated. This means that the intrinsic camera parameters are known. We have the following relation between the two matrices,

$$E = K^\top F K.$$

8-point algorithm. Given that we know 8 point correspondences,[22] we can compute F using direct linear transformation (DLT) derived from

[21] The pseudo-inverse can be computed with SVD as $M^+ = V \frac{1}{\Sigma} U^\top$. $M^+ x$ maps x to some point y that mapped to $x = My$. However, this is not necessarily y, since we lose one dimension of information in the 3×4 matrix P.

The cross product matrix is the conversion of a vector to a matrix that would be equivalent to performing a cross product with that vector,

$$[a]_\times \doteq \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix}.$$

F has 7 degrees of freedom, because it is a 3×3 matrix, is defined up to scale, and $\det(F) = 0$ (rank 2).

Note that these equations are not used for estimating the fundamental matrix, since we do not have access to the camera matrices.

E has 5 degrees of freedom, because it has the same parameters as the fundamental matrix, minus the intrinsic camera parameters.

[22] We assume that all our point correspondences are correct, *i.e.*, they satisfy $x_1^\top F x_2 = 0$.

$$\boldsymbol{x}'^{\top} \boldsymbol{F} \boldsymbol{x} = 0,$$

$$
\begin{bmatrix}
x'_1 x_1 & x'_1 y_1 & x'_1 & y'_1 x_1 & y'_1 y_1 & y'_1 & x_1 & y_1 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x'_n x_n & x'_n y_n & x'_n & y'_n x_n & y'_n y_n & y'_n & x_n & y_n & 1
\end{bmatrix}
\begin{bmatrix}
f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33}
\end{bmatrix} = \boldsymbol{0},
$$

Let A denote the matrix, and f the vector of fundamental matrix entries.

after which we use the SVD trick to find the best algebraic solution.

However, the problem with this is that the values of the matrix grow very large, because of the multiplications, while the last column vector remains 1. Using DLT, this will always result in the best matrix being very similar to the following,

$$
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}.
$$

Thus, importantly, we need to first normalize the coordinates of the images to be in $[-1,1] \times [-1,1]$ before computing $\boldsymbol{F}$.

Now, we also need to enforce the *singularity constraint* (rank 2, $\det(\boldsymbol{F}) = 0$) of $\boldsymbol{F}$, since DLT will give a rank 3 matrix. One possible solution is to set the smallest singular value to equal 0 using SVD. However, this does not necessarily yield the best results, because we are adjusting it a posteriori.

7-point algorithm.   We would like our algorithm to only be able to give rank-2 matrices. We could use 7 point-correspondences, making A a 7×9 matrix.[23] Thus, this matrix has a 2-dimensional null space, which we denote by f_1 and f_2 (with corresponding fundamental matrices F_1 and F_2). This means that we have a family of solutions to the problem such that $A(f_1 + \lambda f_2) = \boldsymbol{0}$,

[23] We could also use more and then use the space corresponding to the two smallest singular values

$$F_1 + \lambda F_2.$$

However, this does not necessarily give a rank-2 matrix. The solution to this problem satisfies the following,

$$\det(F_1 + \lambda F_2) = 0,$$

which results in a cubic equation with 1 or 3 solutions for λ.

5-point algorithm.   The essential matrix only has 5 degrees of freedom, thus we only need 5 point correspondences to compute it. However, as in the 7-point algorithm, this results in a 4-dimensional null space. Furthermore, we additionally have the constraint that E has two equal non-zero singular values. This results in 10 cubic polynomials that need to be solved for.

10.2   Camera geometry

From the essential matrix,[24] we can compute the relative pose between the 2 cameras [Longuet-Higgins, 1981]. This is done by first decomposing $E = U \Sigma V^\top$.[25] Then, we can compute the four possible relative poses as the following,

$$T = U_3, \quad R = UWV^\top$$
$$T = -U_3, \quad R = UWV^\top$$
$$T = U_3, \quad R = UW^\top V^\top$$
$$T = -U_3, \quad R = UW^\top V^\top,$$

where W is the following matrix,

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For each configuration, we then check for the *Cheirality condition, i.e.,* the reconstructed points must be in front of the cameras.

10.3   Scene geometry

From the relative camera pose and point correspondences, we can triangulate the points to find them in three dimensions. A possible solution would be to backproject the x_1 and x_2 and find their intersection.

We can do this by first finding the x and y lines that the points x_1 and x_2 are within their respective image. Let x be the point in the image, then we have the following relations,

$$x = \ell_x \times \ell_y$$
$$\ell_x = \begin{bmatrix} -1 & 0 & x \end{bmatrix}^\top$$
$$\ell_y = \begin{bmatrix} 0 & -1 & y \end{bmatrix}^\top.$$

Then, we can find the planes on which these lines are within the 3-dimensional space,

$$\Pi_x = P^\top \ell_x$$
$$\Pi_y = P^\top \ell_y.$$

Lastly, to get x's backprojection, we have to find the intersection between these planes, which is defined by the points X that satisfy the following,

$$\begin{bmatrix} \Pi_x^\top \\ \Pi_y^\top \end{bmatrix} X = 0.$$

We want to find the intersection of the two backprojections of x_1 and x_2.[26] We can find this by finding a solution for the two backprojections
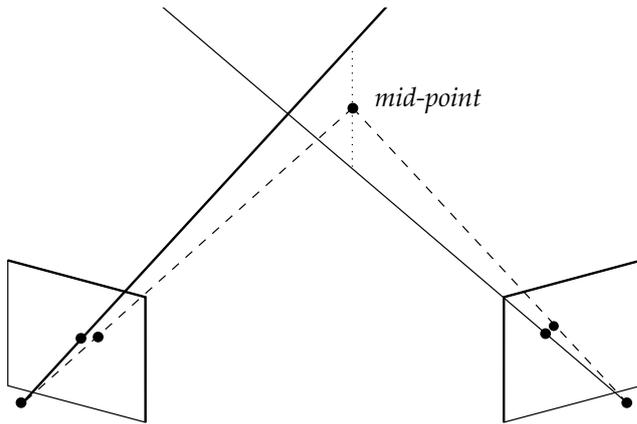
Figure 10.3. Mid-point reprojection. Since the rays will likely never intersect, we find the segment with the smallest length between the two rays. Then, we take the mid-point of this segment as the point. By reprojection the mid-point onto the images, we can compute the reprojection error.

that satisfies the following,

$$\det\left(\begin{bmatrix}\boldsymbol{\ell}_{x_1}^\top \boldsymbol{P}_1 \\ \boldsymbol{\ell}_{y_1}^\top \boldsymbol{P}_1 \\ \boldsymbol{\ell}_{x_2}^\top \boldsymbol{P}_2 \\ \boldsymbol{\ell}_{y_2}^\top \boldsymbol{P}_2\end{bmatrix}\right) = 0.$$

However, due to noise, the two backprojections will never truly intersect. Thus, we need to find a best intersection. A natural best estimate is the mid-point of the shortest line between the two backprojection lines. This minimizes the 3-dimensional error, but not the reprojection error. We could further optimize this estimate to minimize the reprojection error,

$$d(\boldsymbol{x}_1, \boldsymbol{P}_1 \hat{\boldsymbol{X}})^2 + d(\boldsymbol{x}_2, \boldsymbol{P}_2 \hat{\boldsymbol{X}})^2.$$

11   Structure from motion

In *Structure from Motion* (SfM), we want to recover the 3-dimensional structure from m images that depict n 3-dimensional points. Recovering the structure involves estimating the camera matrices, $P_1, \ldots, P_m$, and the positions of the points $X_1, \ldots, X_n$. The fact that we have more than 2 viewpoints enables us to reveal and remove more mismatches.
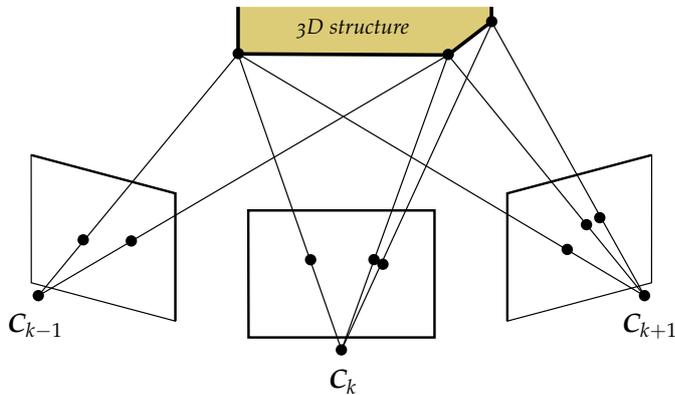
Figure 11.1. Structure from motion. We iteratively extend and refine the recovered structure.

In *sequential SfM*, we initialize the 3-dimensional structure from two images, using the same technique as in the previous section. Then, for each additional view, make a reconstruction with all previous views. In the new view, there are old points that we had already reconstructed, but also new points that are new to the reconstruction. We refine the old points by considering the new view, and we extend the structure by adding the new points.

Afterward, we refine the recovered structure. We use *bundle adjustment*, which is a non-linear method that minimizes the sum of squared reprojection errors,

$$\sum_{i=1}^{m} \sum_{j=1}^{n} d\left(x_{ij}, P_i \hat{X}_j\right)^2.$$

In general, bundle adjustment involves a large amount of parameters. However, there are many methods for solving this problem efficiently.

11.1   Robust estimation

When optimizing the geometric error when backprojecting, the squared error can be a source of bias for outliers. One possible solution is to use an M-estimator that gives more weight to small errors and less weight to large errors. For example, we can use the robust norm function ρ to give outliers less influence over the fit,

$$\rho(\epsilon; \sigma) = \frac{\epsilon^2}{\sigma^2 + \epsilon^2}.$$

Choosing the correct scale σ is critical. A popular choice is 1.4826, which is plotted in Figure 11.2.
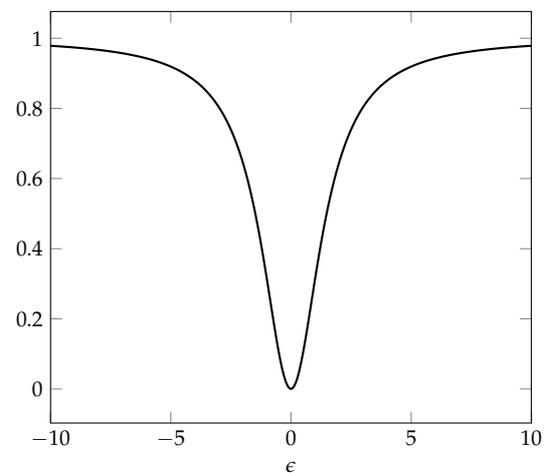
Figure 11.2. Robust norm function $\rho(\epsilon; \sigma)$ with $\sigma = 1.4826$.

Another solution is to use the *random sample consensus* (RANSAC) algorithm to robustly compute the best fit. For k iterations, the algorithm computes a minimum model with n data points. It then checks how many "inliers" this model has, which, intuitively, means how many points does this model explain well enough. If the amount of inliers is above the threshold d, we fit the model to all inliers, and compute the error on all points. We return the model with the lowest error on its inliers.

```
function RANSAC(𝒳,n,k,t,d)
    ε_min ← ∞
    initialize ℳ_best
    repeat k times
        𝒬 ← n points randomly sampled from 𝒳
        ℳ ← FitModel(𝒬)
        ℛ ← Residual(𝒳, ℳ, Error)
        ℐ ← {x ∈ 𝒳 | ℛ(x) ≤ t}
        if |ℐ| ≥ d then
            ℳ_inliers ← FitModel(ℐ)
            ε ← Error(ℳ_inliers, ℐ)
            if ε < ε_min then
                ℳ_best ← ℳ
            end if
        end if
    endreturn ℳ_best
end function
```

Algorithm 1. Random sample consensus algorithm. $\mathcal{X}$ are the data points, n is the minimum number of data points needed to fit a model, k is the amount of iterations to run, t is the upper bound on the error for considering a point as an inlier, and d is the minimum amount of inliers to consider a model "good".

Let w be the probability of a point being an in inlier. Then, w^n is the probability that all n points used to fit the model are inliers. $1 - w^n$ is the probability that at least one of the points is not an inlier, which would result in a bad model. $(1 - w^n)^k$ is the probability that RANSAC does not select a single subset with all inliers. If we let p be the probability that RANSAC will return a good model, then we have the following identity,

$$1 - p = (1 - w^n)^k,$$

which means that if we want probability p to get a good model, then we can compute the necessary k as follows,

$$k = \frac{\log(1 - p)}{\log(1 - w^n)}.$$

12   *Stereo matching*

In humans and machines, depth perception is very important to interact with the environment. In humans, this is done by two eyes that are slightly offset. From this difference, depth can be estimated. We can do the same with cameras with potentially even more views, which could result in better depth estimation.

The standard setup in stereo geometry is two identical cameras that are only displaced on the x-axis. In other words, they have the same rotation and height. Then, it is easy to compute the fundamental matrix,

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}.$$

Then, we have the following epipolar lines,

$$\ell_1 = x_2^\top F = \begin{bmatrix} 0 & 1 & -y_2 \end{bmatrix}^\top$$

$$\ell_2 = F x_1 = \begin{bmatrix} 0 & 1 & -y_1 \end{bmatrix}^\top,$$

which tells us that all points will have the same y-axis, and are parallel to the x-axis.

If there is a 3-dimensional point at $X = [X, Y, Z]$, then we have the following relationships with the 2-dimensional points on the image planes,

$$x_1 = f \frac{X}{Z}$$

$$y_1 = f \frac{Y}{Z}$$

$$x_2 = f \frac{X - b_x}{Z}$$

$$y_2 = f \frac{Y}{Z}.$$

The *stereo disparity* (which is the difference in pixel locations) $d = x_1 - x_2$ can then be computed as follows,

$$d = x_1 - x_2$$

$$= f \frac{X}{Z} - f \frac{X - b_x}{Z}$$

$$= f \frac{b_x}{Z}$$

We can then solve for the depth Z,

$$Z = \frac{f b_x}{d}$$

Furthermore, we can compute how much the disparity changes as the depth changes,

$$\frac{\mathrm{d}d}{\mathrm{d}Z} = \frac{\mathrm{d}}{\mathrm{d}Z} \frac{f b_x}{Z}$$

$$= \frac{f b_x}{Z^2},$$

If the cameras are not perfectly on the same x-axis or do not have the same rotation, we can solve it by rectifying them using a homography [Loop and Zhang, 1999]. Geometrically, this means first projecting the two images onto a common plane that is parallel to the baseline. It can also be seen as rectifying the epipolar lines.

Under the assumption that camera 1 is at the origin w.r.t. camera 2.

Translating camera to the left is the same as translating the world to the right. b_x is the baseline on the x-axis.
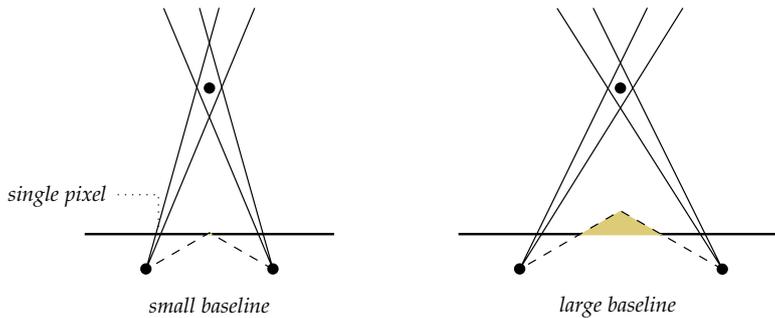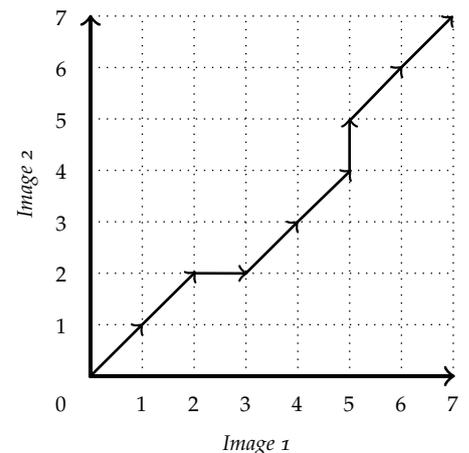
which tells us that the uncertainty in the pixel location grows quadratically with depth, causing large depth error. But, this uncertainty is also inversely proportional to the baseline, thus we can increase b_x to see depth better at deep locations. However, this comes at the cost that the disparity becomes larger as the baseline increases, which makes the search problem of matching the pixels between images harder. Furthermore, a large baseline results in more occlusion problems, where one camera is occluded while the other is not, making it impossible to estimate depth.[27] If we have a small baseline, the disparity is low, which makes it harder to achieve fine-grained resolution, since pixels are not infinitely small. See Figure 12.1 for an illustration of this.

[27] A small baseline makes single-sided occlusion less common, because they are more likely to both be occluded, which is not a problem, since we then estimate the depth to the occlusion.

Now, we need to figure out how to match the pixels in one image to another. Note that the search is limited to the epipolar line, so we only need to match along the common epipolar line of the two image planes. We also need the points to enforce spatial consistency, since close points in the environment are also close between the image planes. Also, we can never turn back, *i.e.*, if x_1 is matched to x_2, then $x_1' > x_1$ cannot match to a point $x_2' < x_2$. We then associate with each matching of pixels a cost that is computed from their intensities. We can then use a matching algorithm to compute the optimal matching [Baker, 1981].

12.1   Multi-view stereo

We can also add more cameras to the setup. An obvious advantage is that we can set the cameras up such that we have multiple baselines [Okutomi and Kanade, 1993]. This has the result that we no longer have the trade-off between large and small baseline, because we can use both. This makes it possible to compute the depth of far points accurately and deal with the visibility problem of large baselines. An example of a multi-view setup being more robust to occlusions is Figure 12.3. The left two cameras would not be able to find the depth of the point on the cube, but the introduction of the right camera made it possible.

Figure 12.2. Matching algorithm, where the points on the corresponding lines are being matched to each other. A line to the right means that there is an obstruction.
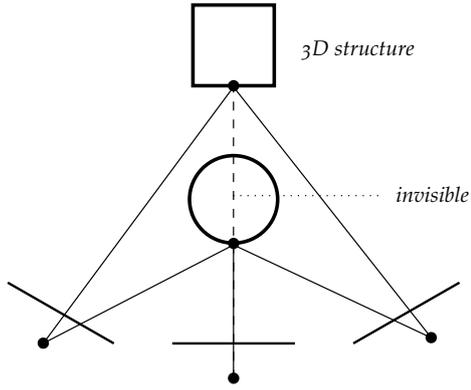
12.2   Volumetric stereo

In volumetric stereo, we discretize the 3-dimensional space into voxels.[28] We then "carve" the geometry out of these voxels. This gives us a reconstruction of the scene, from which we can create a depth map by taking the first occupied voxel for each pixel. We assign a photoconsistency cost to each voxel, and then we make the minimum cost cut in this graph.

We can also use *visibility-first* methods. In these methods, we first handle the visibility problem, and then we carve out the voxels. We do this by first segmenting the foreground from the background for every camera. We then backproject the foreground silhouette to the image plane. All these pixels are visible from this camera. The visual hull is then the intersection of these silhouettes. We only keep the voxels that are part of the visual hull, which gives us a tight bound on the object.

[28] A voxel is a "3-dimensional pixel".
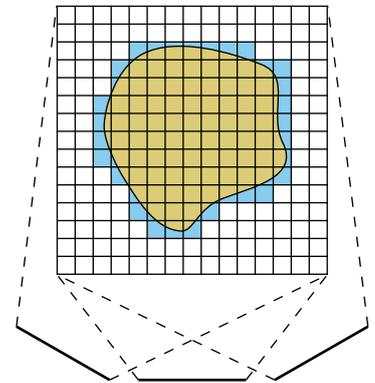
Figure 12.4. Above view of voxel carving.
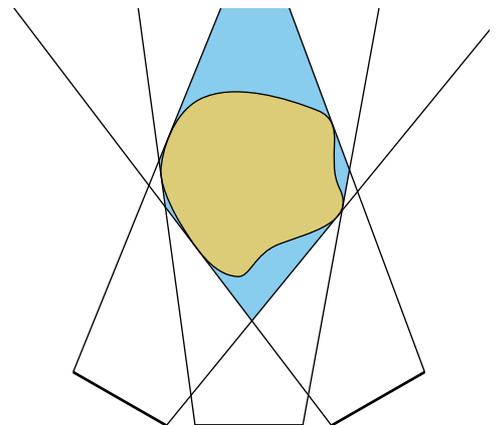
Figure 12.5. The visual hull of the object with three cameras. As we increase the amount of cameras, the intersection becomes smaller and thus the bound on the object tighter.

References

Henry Harlyn Baker. *Depth from edge and intensity based stereo*. University of Illinois at Urbana-Champaign, 1981.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *icml*, volume 96, pages 148–156. Bari, Italy, 1996.

Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Manchester, UK, 1988.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

H Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133–135, 1981.

Charles Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 1, pages 125–131. IEEE, 1999.

David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

Masatoshi Okutomi and Takeo Kanade. A multiple-baseline stereo. *IEEE Transactions on pattern analysis and machine intelligence*, 15(4):353–363, 1993.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the*

IEEE conference on computer vision and pattern recognition, pages 779–788, 2016.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning internal representations by error propagation, 1985.

Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104:154–171, 2013.